Objectives

- Code Smells
- Refactoring

GitHub: `main` branch instead of `master` branch

1

Review

1. What is guaranteed in software development?
   - This informs how we design our code
2. What are some of the best practices in object-oriented design?
   - Provide an example of the practice (in our assignments, in our discussions, in Java, …)
3. What is refactoring?
4. What is the process for writing maintainable code?

2

Review: Designing Systems

All systems **change** during their life cycle

- Questions to consider:
  - How can we create designs that are stable in the face of change?
  - How do we know if our designs aren't maintainable?
  - What can we do if our code isn't maintainable?
- Answers will help us
  - Design our own code
  - Understand others' code

3

Review: Overview Best Practices

- (DRY): Don't repeat yourself
- Single Responsibility Principle
- Shy
  - Avoid Coupling
- Tell, Don't Ask
- Open-closed principle
- Avoid code smells

A lot of similar, related fundamental principles

4

Review: Process to Write Maintainable Code

- Apply the design principles, but as your code evolves, you'll see that you didn't always adhere to them

1. Identify code smell

2. ***Refactor*** code to remove code smell
3. ***Test*** code to confirm code still works

Review: Code Smells

A hint in the code that something could be designed better

- Duplicated code
- Long method
- Large class
- Long parameter list
- Very similar child classes
- Too many public variables
- Empty catch clauses
- Switch statements/long if statements
- Shotgun surgery
- Literals
- Global variables
- Side effects
- Using `instanceof`

Code Smells

- For each of the following code smells, state
  - Why these may occur in code
  - Why they are a problem in terms of maintaining code
  - How to fix them
- Code smells:
  - Long methods
  - Large class
  - Magic numbers (e.g., -1 or 480 in code)
  - Comments (non-API/Javadoc comments)

7

Code Smell: Long Methods

- What's the problem with long methods?
- What made us write them?
- How can we fix them?
- What is an issue with lots of short methods?

8

Long Methods: Issues and Solutions

- Issues:
  - Hard to understand (see) what method does
  - Smaller methods have reader overhead
    - Look at code for called methods
    - But, should use descriptive names
    - In Eclipse, use F3 to jump to a method's definition
- Solutions:
  - Find lines of code that go together (may be identified by a comment) and extract method

Code Smell: Large Class

- What's the problem?

Large Class

- Issue: Too many instance variables → trying to do too much
  - Violates **Single Responsibility Principle**
- Solutions:
  - Bundle groups of variables together into another class
    - Look for common prefixes or suffixes
  - If includes optional instance variables (only sometimes used), create child classes
  - Look at how users use the class for ideas of how to break it up

Eclipse: Refactor → Extract Class or Extract Superclass

11

Literals or Magic Numbers

- If a number has a special meaning, make it a constant
- Example: Distinguish between 0 and NO_VALUE_ASSIGNED
  - If value changes (e.g., -1 instead of 0), only one place to change
  - Less error-prone (e.g., was I using 1 or -1?)

Eclipse: Refactor → Extract Constant

12

Comments

Problem: Comments used as Febreze to cover up smells

- Describe what the code or method is doing
- Should be reserved for *why*, not what

- Solutions:
  - If need a comment for a block of code (or a long statement) → create a method with a descriptive name
  - If need a comment to describe method, rename method with more descriptive name

These [internal] comments are different from API comments

Code Smell: Using `instanceof`

```
public void drawShape( Shape shape ) {
    if ( shape instanceof Square ) {
        drawSquare(shape);
    }
    else if( shape instanceof Circle ) {
        drawCircle(shape);
    }
}
```

- Why isn't this good code?
  - Always consider: how is this code likely to change?
- How could we write this in a better way?

Code Smell: Using `instanceof`

- Previous example: had to know all of the Shape classes
  - Update whenever a Shape is added or removed
- Better code: ***Polymorphic***!

```
public void drawShape( Shape shape ) {
    shape.draw();
}
```


Lazy Class

- Problem
  - Class in question doesn't do much
  - Classes cost time and money to maintain and understand
- How could this happen?
  - Refactoring!
  - Planned to be implemented but never happened
- Solution
  - Get rid of class
    - Inline or collapse subclass into parent class

Speculative Generality

- Beware of too much abstraction, allowing for too much flexibility that isn't required
- Solution: Collapse classes

22

More Code Smells

- Discuss more code smells and solutions (Design Patterns) later

23

Software Design Rules of Thumb

- Code smells are not ***always*** bad
  - Do not always mean code is poorly designed
- Open code is not ***always*** bad
- Need to use your judgment
  - Good judgment comes from experience.
  - How do you get experience? *Bad judgment works every time*

Goal: Gain experience to improve your judgment

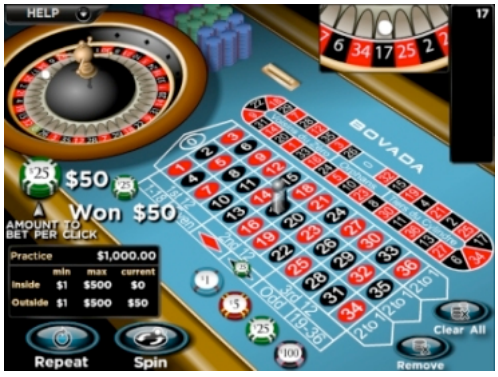
Refactoring Summary

- Write code and then *rewrite* code
  - Eye toward extensibility, flexibility, maintainability, and readability
  - Maintain correctness
- Reading/understanding other people's code can be difficult
  - Make your code readable, understandable
- Probably impossible to design/write "correctly" the first time
  - A lot harder to get the logic right, make sure you're not creating bugs, know/check the right answer…
  - Don't necessarily know what is likely to change

Readability, Maintainability, Extensibility

REFACTORING PRACTICE

Simulating a Roulette Game



Understanding Code

- Execute the code
  - What is the main driver for this project?
- What are each class's responsibilities?

28

Bug in the Code

- Determining if Odd/Even Bet was won is incorrect

29

Understanding Code

- Focus: how **open** is the code to adding **new** kinds of **bets** and how **closed** it is to **modification**?
  - How many classes know about the `Bet` class?
  - What code would need to be added to `Game` to allow the user to make another kind of bet that paid one to one odds and was based on whether the number spun was high (between 19 and 36) or low (between 1 and 18)?

Roulette

- Goals
  - Learn to read, understand someone else's code
    - Refactoring can help
  - Refactor for readability
  - Justify decisions
- No "right" answer
  - Many design decisions
  - Defend your design decisions in analysis

TODO: Assignment 8

- Due next Wednesday